A circular logo with text and symbols

Description automatically generated

**CSN 252**

**SYSTEM SOFTWARE**

**TUTORIAL 8**

***DESIGN OF SIC/XE ASSEMBLER***

NAME: Divij Rawal

ENROLLMENT NO.: 22114031

BATCH: O2

**INTRODUCTION**

Constructed using the C++ language, this assembler demonstrates proficiency in processing SIC/XE programs, showcasing advanced parsing capabilities and robust functionality. Operating in a dual-pass system, this assembler, adeptly accommodates all four addressing modes (1, 2, 3, and 4), showcasing comprehensive functionality and precision. Following features are implemented in this assembler:

1. Expressions
2. Literals
3. Symbols and Statements
4. Control Section

This assembler works in two passes:

* Pass 1:
  + Creates a "symbol table" to track labels (names) used in your code.
  + Generates a temporary "intermediate file" containing details about each line of assembly code.
  + Find “errors” in format if possible.
* Pass 2:
  + Produces a "listing file" showing your original code with its memory location (address), a code section identifier (block number), and the translated machine instructions (object code).
  + Generates the final executable file (object code file) for the computer to run.
  + Creates an "error file" if any issues are found in your code, helping you identify the problems.

**STEPS TO COMBINE**

1. Go into folder:

**cd /SICXE-ASSEMBLER/**

1. Compile all the files into one:

**g++ Main.cpp IO.cpp ExpressionEvaluator.cpp Pass1.cpp OpTab.cpp StringUtilities.cpp Pass2.cpp Validate.cpp OperandHandler.cpp -o Assembler**

1. Run the file:

**. /Assembler input.txt**

1. Output will be in listFile.txt and objectFile.txt and error will be logged in console.

**CAUTIONS**

1. Try to give proper indentation as possible for better results.
2. **Do not use numbers while defining symbol names.**

**DATA STRUCTURES**

1. OpInfo: Holds information about an operation mnemonic, its format, and opcode.
2. parsedLine: Represents a parsed line of assembly code with label, opcode, operands, and error message.
3. opCode: Contains information about an operation, including its opcode, displacement, flags, format, and registers.
4. symInfo: Stores information about a symbol, such as its address and relocatability.
5. toWrite: Holds data to be written, along with a type of indicator for error, comment, or formatted lines.
6. modification: Represents modification records, including address, number of bytes, external reference flag, signs, and references.
7. locatedParsedLine: Combines a parsed line with its location counter value.
8. cSect: Represents a control section, containing name, location counter, external references, external definitions, literal table, data section, parsed lines with location, modification records, and symbol table.
9. opTab: Represents optable.
10. litTab: Represents literal table.

**FUNCTIONS**

**Pass1**

1. handelBYTE: Handles processing of BYTE directives, validating and extracting data from hexadecimal or character strings.
2. addLableToSymTab: Adds a label to the symbol table of a given control section, checking for duplicate definitions.
3. writeFomatted: Writes a formatted line to the current control section's data section.
4. incLocCtr: Increments the location counter of the current control section by a specified increment and checks for address range overflow.
5. writeAndIncr: Writes a formatted line and increments the location counter, handling different scenarios based on program start and end.
6. hasExternalReference: Checks if a line has an external reference and extracts references along with their signs for modification records.
7. handelModRecord: Handles modification records based on whether the instruction has a format 4 or uses external references.
8. checkLabels: Checks if labels in an instruction are defined within the symbol table of the current control section.
9. handleEquate: Handles EQU directives by evaluating expressions and adding the label and its value to the symbol table.
10. handleOrg: Handles ORG directives by evaluating expressions and updating the location counter accordingly.
11. isLit: Checks if an operand is a literal.
12. literalLength: Calculates the length of a literal based on its type and content.
13. writeLiterals: Writes literals to the current control section and updates the location counter.
14. addExtRefs: Adds external references to the current control section.
15. addExtDefs: Adds external definitions to the current control section.
16. getlocCtrIncr: Processes a parsed line, determines the increment to the location counter, and handles various directives and operations.
17. isComment: Checks if a parsed line is a comment.
18. isEmptyLine: Checks if a parsed line is empty.
19. runPass1: Runs the first pass of the assembler, processing a vector of parsed lines, updating the current control section, and handling errors.

**Expression** **Evaluator**

1. setCSECT: Sets the current control section for expression evaluation.
2. replace: Replaces occurrences of a substring in a string with another substring.
3. replaceAll: Replaces all occurrences of a substring in a string with another substring.
4. isValidNumber: Checks if a string represents a valid number.
5. isValidOpName: Checks if a string represents a valid operation name.
6. isOperator: Checks if a character is an operator.
7. valueOf: Gets the value of a symbol from the symbol table or external references.
8. substitute: Substitutes symbols in an expression with their respective values.
9. comparePeriority: Compares the priority of two operators.
10. evaluateSimpleExp: Evaluates a simple expression consisting of two operands and an operator.
11. resultingRelativityDegree: Determines the relativity degree of the result of an operation on relative addresses.
12. valueOfExplicitExp: Evaluates an explicit expression containing multiple operands and operators.
13. validate: Validates an expression and determines if operands are relative or absolute.
14. evaluateExpression: Evaluates an expression using the current control section.
15. validateExpression: Validates if an expression is syntactically correct.
16. isRelativeExpression: Determines if an expression evaluates to a relative address.

**Pass2**

1. getRegMap: Returns a map containing register names as keys and their corresponding numerical values as values.
2. generateOpCode: Generates the opcode for a given parsed assembly line (parsedLine). It determines the instruction format and computes the displacement based on the addressing mode and operand values.
3. calcOpCode: Converts the generated opCode struct into a string representation of the opcode.
4. binaryToDec: Converts a binary representation of flags into decimal values for NI (n and i) and XBPE (x, b, p, and e) fields.

runPass2: function orchestrates the execution of Pass 2. It takes an output file stream pointer (opFile) and a vector of control sections (cSects) as input. It initializes error messages and sets up the header for the output file. Then, it iterates through each control section, generating object codes and writing corresponding lines to the output file. It also handles error messages and writes them to the output file if encountered. After processing all control sections, it finalizes the output file by writing header records, define records, reference records, text records, modification records, and borders for each control section.

**Validator**

* validate: This function validates an assembly language instruction represented by a parsedLine object. It checks for errors in the label, opcode, and operands based on predefined rules. It also distinguishes between instructions, directives, and reserved words, and delegates validation accordingly.
* f1, f2, f34: These functions validate instructions with different formats (1, 2, 3/4) and their operands. They throw exceptions if the instruction format or operands are invalid.
* isReg: This function checks if a string represents a valid register name.
* fillDirectives: This function initializes the directives map with assembly language directives and their corresponding characters.
* checkBYTE, checkLiteral, isChar, isRubbish, isNumber, checkExt: These functions handle specific validations related to byte strings, literals, characters, numbers, and external references.

**String Utilities**

1. intToString: Converts an integer to a string. Optionally, it can convert the integer to its hexadecimal representation.
2. adjustStringLength: Adjusts the length of a string by adding a specified character either at the beginning (left-hand side) or at the end (right-hand side) until it reaches the desired length.
3. toUpperCase: Converts a string to uppercase.
4. trim: Removes leading and trailing spaces from a string.
5. parseDecimal: Parses a decimal integer from a string.
6. parseHexaDecimal: Parses a hexadecimal integer from a string.
7. parseFloat: Parses a floating-point number from a string.
8. split: Splits a string into substrings based on a delimiter character.
9. mergeStrings: Merges two strings with a specified delimiter character between them.
10. isNumeric: Checks if a string represents a numeric value.
11. isHexa: Checks if a string represents a hexadecimal value.
12. generalSplit: Splits a string into substrings based on a set of delimiter characters

**IO for Files:**

1. constructLine: Constructs a line of code with specified formatting, including line number, address, label, operation, and operands.
2. constructLine\_Pass2: Similar to constructLine, but also includes fields for flags and opcode for Pass 2.
3. deleteFile: Deletes a file with the specified name.
4. writeLine: Writes a line of text to the specified output file stream.
5. readLine: Reads a line of text from the specified input file stream.
6. writeHeader: Writes a header to the output file stream for Pass 1.
7. writeHeader\_Pass2: Writes a header to the output file stream for Pass 2.
8. writeError: Writes an error message to the output file stream.
9. writeBorder: Writes a border to the output file stream.
10. writeBorder\_Pass2: Writes a border to the output file stream for Pass 2.
11. writeSymTab: Writes the symbol table to the output file stream.
12. writeComment: Writes a comment to the output file stream.
13. writeHeaderRecord: Writes a header record to the output file stream.
14. writeTextRecords: Writes text records to the output file stream.
15. writeModRecords: Writes modification records to the output file stream.
16. writeDefineRecord: Writes a define record to the output file stream.
17. writeReferRecord: Writes a reference record to the output file stream.
18. writeEndRecord: Placeholder function for writing the end record.

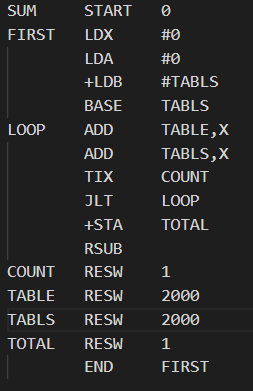
**Main**

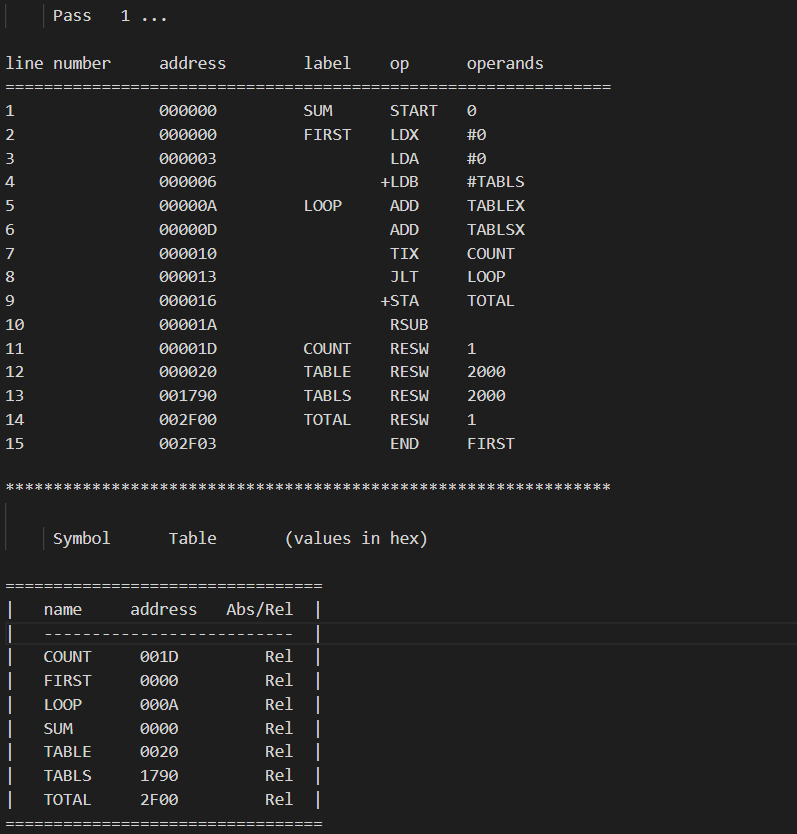
1. needNoOperands: Checks if an opcode requires no operands by comparing it against a list of specific opcodes.
2. parse: Parses a line of assembly code, extracting label, opcode, and operands into a struct.
3. toUppercase: Converts a string to uppercase for uniformity in comparison.

**SAMPLE PROGRAM**

* 1. **Question 3 of Section 2.2 from LL Beck**

**TEXT**



**LISTING** **FILE**A screenshot of a computer

Description automatically generated

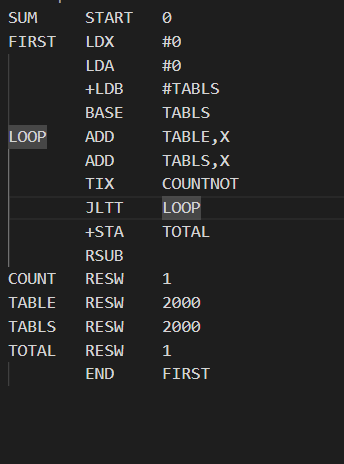
**OBJECT** **FILE**

A computer screen with white text

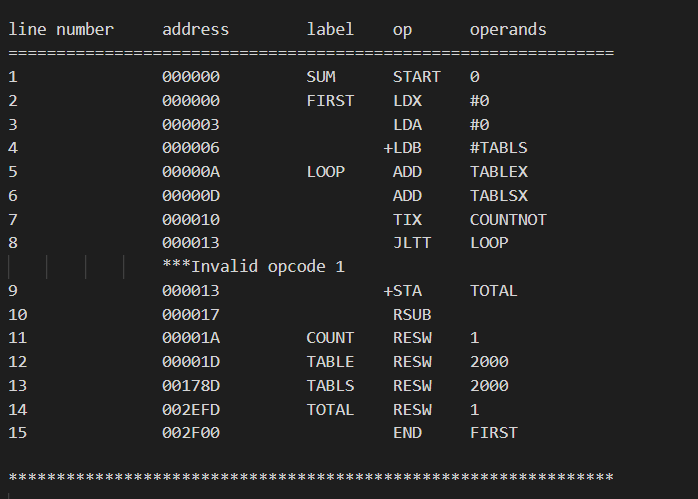
Description automatically generated

* 1. **SAMPLE INCORRECT PROGRAM**

**TEXT FILE**



**LISTING FILE**

****